MOSES: Supporting and Enforcing Security Profiles on Smartphones

Yury Zhauniarovich, Giovanni Russello, *Member, IEEE*, Mauro Conti, *Member, IEEE*, Bruno Crispo, *Senior Member, IEEE*, and Earlence Fernandes, *Member, IEEE*

Abstract—Smartphones are very effective tools for increasing the productivity of business users. With their increasing computational power and storage capacity, smartphones allow end users to perform several tasks and be always updated while on the move. Companies are willing to support employee-owned smartphones because of the increase in productivity of their employees. However, security concerns about data sharing, leakage and loss have hindered the adoption of smartphones for corporate use. In this paper we present MOSES, a policy-based framework for enforcing software isolation of applications and data on the Android platform. In MOSES, it is possible to define distinct *Security Profiles* within a single smartphone. Each security profile is associated with a set of policies that control the access to applications and data. Profiles are not predefined or hardcoded, they can be specified and applied at any time. One of the main characteristics of MOSES is the dynamic switching from one security profile to another. We run a thorough set of experiments using our full implementation of MOSES. The results of the experiments confirm the feasibility of our proposal.

Index Terms—Android, BYOD, virtualization, access control, context

1 Introduction

Worldwide smartphone sales totalled 250 million units in the third quarter of 2013, up 46 percent from the same quarter of 2012 [1]. In the smartphone domain, the Android OS is by far the most popular platform with 82 percent market share. Those figures clearly show the pervasiveness of Android, mostly justified by its openness to third party developers.

Smartphones allow end users to perform several tasks while being on the move. As a consequence, end users require their personal smartphones to be connected to their work IT infrastructure. More and more companies nowadays provide mobile versions of their desktop applications. Studies have shown that allowing access to enterprise services with smartphones increases employee productivity [2]. An increasing number of companies are even embracing the BYOD: Bring Your Own Device policy [3], leveraging the employee's smartphone to provide mobile access to company's applications. Several device manufacturers are even following this trend by producing smartphones able to handle two subscriber identification modules (SIMs) at the same time.

Despite this positive scenario, since users can install third-party applications on their smartphones, several security concerns may arise. For instance, malicious applications may access emails, SMS and MMS stored in the smartphone containing company confidential data. Even more worrying is the number of *legitimate* applications harvesting and leaking data that are not strictly necessary for the functions the applications advertise to users [4], [5]. This poses serious security concerns to sensitive corporate data, especially when the standard security mechanisms offered by the platform are not sufficient to protect the users from such attacks.

One possible solution to this problem is *isolation*, by keeping applications and data related to work separated from recreational applications and private/personal data. Within the same device, separate *security environments* might exist: one security environment could be only restricted to sensitive/corporate data and trusted applications; a second security environment could be used for entertainment where third-party games and popular applications could be installed. As long as applications from the second environment are not able to access data of the first environment the risk of leakage of sensitive information can be greatly reduced.

Such a solution could be implemented by means of virtualization technologies where different instances of an OS can run separately on the same device. Although virtualization is quite effective when deployed in full-fledged devices (PC and servers), it is still too resource demanding for embedded systems such as smartphones. Another approach that is less resource demanding is paravirtualization. Unlikely full virtualization where the guest OS is not aware of running in a virtualised environment, in paravirtualization it is necessary to modify the guest OS to boost performance. Paravirtualization for smartphones is currently under development and several solutions exist (e.g., Trango, VirtualLogix, L4 microkernel [6], L4Android [7], [8]). However, all the virtualization solutions suffer from having a coarse grained approach (i.e., the virtualised

- *Y. Zhauniarovich and B. Crispo are with the University of Trento, Trento 38123, Italy. E-mail: {yury.zhauniarovich, bruno.crispo}@unitn.it.*
- *G. Russello is with the University of Auckland, Auckland, 1142, New Zeland. E-mail: g.russello@auckland.ac.nz.*
- *M. Conti is with the University of Padua, Padua, 35131, Italy. E-mail: conti@math.unipd.it.*
- *E. Fernandes is with the University of Michigan, Ann Arbor, MI 48109-2121. E-mail: earlence@umich.edu.*

Manuscript received 30 July 2013; revised 3 Dec. 2013; accepted 9 Jan. 2014; date of publication 15 Jan. 2014; date of current version 14 May 2014.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TDSC.2014.2300482

environments are completely separated, even when this might be a limitation for interaction). Other limitation is the hardcoding of the environment specification. Environments cannot be defined by the user/company according to their needs but they are predefined and hardcoded in the virtual machine. Furthermore, the switching among environments always require user interactions and it could take a significant amount of time and power. While researchers are improving some of these aspects [9], the complete separation of virtual machines and the impossibility to change or adapt their specifications remain an open issue.

1.1 Contributions

This paper presents MOSES, a solution for separating modes of use in smartphones. MOSES implements soft virtualization through controlled software isolation. With MOSES:

- Each security profile (SP) can be associated to one or more contexts that determine when the profile become active. Contexts are defined in term of low level features (e.g., time and location) and high level features (reputation, trust level, etc.).
- Both contexts and profiles can be easily and dynamically specified by end users. MOSES provides a GUI for this purpose. Profiles can be fine-grained to the level of single object (e.g., file, SMS) and single application.
- Switching between security profiles can require user interaction or be automatic, efficient, and transparent to the user.

We implemented MOSES and ran a thorough set of experiments to evaluate its efficiency and effectiveness. The experiments show the feasibility and accepted performance of our solution for storage and energy consumption.

The rest of this paper is organised as follows. In Section 2 we recall some preliminaries for the Android security, while we discuss the related work in Section 3. MOSES is presented in Section 4, while details of its architecture are discussed in Section 5. Section 6 covers MOSES implementation in details. In Section 7 we describe a thorough evaluation of MOSES. Concluding section discusses limitations of the current implementation, and envisages possible future work.

2 Android Security

Google Android is a Linux-based mobile platform developed by the Open Handset Alliance (OHA) [10]. Most of the Android applications are programmed in Java and compiled into a custom byte-code that is run by the Dalvik virtual machine (DVM). In particular, each Android package is executed in its own address space and in a separate DVM. Android applications are built combining any of the following four basic components. *Activities* represent a user interface; *Services* execute background processes; *Broadcast Receivers* are mailboxes for communications within components of the same application or belonging to different apps; *Content Providers* store and share application's data. Application components communicate through messages called *Intents*.

Focusing on security, Android combines two levels of enforcement [11], [12]: at the Linux kernel level and the application framework level. At the Linux kernel level Android is a multi-process system. During installation, an application is assigned with a unique Linux user identifier (*UID*) and a group identifier (*GID*). Thus, in the Android OS each application is executed as a different user process within its own isolated address space. All files in the memory of a device are also subject to Linux access control. On a Linux, file access permissions are set for three types of users: the owner of the file, the users who are in the same group with the owner of the file and all other users. For each type a tuple of read, write and execute (r-w-x) permissions is assigned. In Android, by default, the files in the user's home directory can be read, written and executed by the owner and the users from the same group as the owner. All other users cannot work with these files. So as different applications by default have different user identifiers files created by one application cannot be accessed by another.

At the application framework level, Android provides access control through the inter-component communication (ICC) reference monitor. The reference monitor provides mandatory access control (MAC) enforcement on how applications access the components. In the simplest form, protected features are assigned with unique security labels—*permissions*. Protected features may include protected application components and system services (e.g., Bluetooth). To make the use of protected features, the developer of an application must declare the required permissions in its package manifest file: `AndroidManifest.xml`.

As an example, consider an application that needs to monitor incoming SMS messages, `AndroidManifest.xml` included in the application's package would specify: `<uses-permission android:name="android.permission.RECEIVE_SMS"/>`. Permissions declared in the package manifest are granted at the installation time and cannot be modified later. Each permission definition specifies a protection level which can be: `normal` (automatically granted), `dangerous` (requires user confirmation), `signature` (requesting application must be signed with the same key as the application declaring the permission), or `signature or system` (granted to packages signed with the system key or located in the system image).

3 Related Work

This section provides an overview of the related work. In particular, Section 3.1 describes research efforts in enhancing the security of the Android platform. Section 3.2 discusses BYOD approaches for mobile systems. Solutions based on secure container (SC) are presented in Section 3.2.1, while Section 3.2.2 covers virtualization solutions. Section 3.2.3 describes solutions that adopted yet other different approaches.

3.1 Android Security Extensions

There are a lot of solutions proposed to improve the security of Android. We consider the ones that are more related to our system.

In Android, at installation time users grant applications the permissions requested in the manifest file. Android

supports an all-or-nothing approach, meaning that the user has to either grant all the permissions specified in the manifest or abort the installation of the application. Moreover, a permission cannot be revoked at runtime. To circumvent this coarse-grained approach, several solutions have been proposed. Apex [13] allows users to select which permissions to grant to an application during the installation. Saint [14] is a policy-based application management system aiming at controlling how applications interact with each other. CRêPE [15] allows a user to create policies that can automatically control the granting of permissions during runtime.

More recently, [16], [17] concentrate on the protection of the user's private data. In particular, MockDroid [16] is a system that can limit the access of the installed applications to phone data by filtering out information. For instance, an application querying the contacts' provider may receive no results even if the provider is not empty. This approach is further refined in TISSA [17] where users are able to define the accuracy level of the information revealed to the application by means of privacy levels.

Taintdroid [4] proposes dynamic taint analysis to control how data flow between applications. In Taintdroid, taints are statically associated with predefined data sources, such as the contact book, SMS messages, the device identifier (IMEI), etc. Taintdroid tracks the flow of tainted data and notifies the user if the tainted data leave the device through the outbound network connections. By using Taintdroid's tainting capability, AppFence [18] provides additional mechanisms to shadow sensitive data and to block unauthorised leakage of data via network. YAASE [19] encompasses tainting to prevent confuse deputy and privilege escalation attacks. In [20], [21] Taintdroid capabilities are used to enforce data-driven usage control. In [22], [23] taint tracking enables the system to trace sensitive information, enterprise and health data respectively, and enforce policies for that data. Unfortunately, Taintdroid has some limitations such as inability to trace implicit flows. Moreover, it prevents the load of shared libraries by third-party applications to prevent leakages through native code.

Context information plays a pivotal role to enhance security in mobile devices. In [13], [15], [24], [25], context is used to trigger security rules at runtime. The approaches in [20], [21] use context to limit access to data in some environments. In [23], special context is a necessary condition to generate security notifications. In [22] the context is used to taint data generated in predefined environments. FlaskDroid [25] uses context to set up the values of one or more boolean variables in policies. These boolean variables are later used to instantiate a policy that is enforced by FlaskDroid's policy enforcement system, which is based on the extension of SEAndroid mandatory access control. The mandatory access control implemented in [25], [26] considerably diminishes the effect of root exploits.

3.2 Bring Your Own Device Approaches

Besides approaches to improve Android security in general, some solutions specifically aimed at supporting the BYOD have been proposed. The most important are listed below.

3.2.1 Secure Container

Secure container is a special mobile client application that creates an isolated environment on the phone at the application layer. The application allows an enterprise administrator to create policies which control this isolated environment but cannot control the behaviour of a user outside this container [27]. This approach does not require the modification of the system image and is widely explored in the research community. AppGuard system [28] for instance, is a standalone Java application that disassembles apk file, inlines security checks before dangerous instructions according to a selected policy and then reassembles and signs the package. Thus at runtime, before executing a dangerous instruction AppGuard performs a security check and if the instruction is not allowed according to the policy an exception is thrown. Jeon et al. [29] use package rewriting to substitute dangerous instructions to equivalent ones, which are guarded by additional security checks. These guarded functions are implemented in a standalone Android service, which performs the additional checks. Aurasium system [30] intercepts some critical Bionic libc functions (e.g., `read()`, `write()`, `open()`) and calls the Aurasium (safe) version of them.

Many commercial solutions use the concept of security container implemented as a user application. NitroDesk TouchDown [31] and Good [32] offer solution with a prefixed set of business functionality in the container (i.e., email). Other solutions, (for instance, Fixmo [33]) offer a set of basic applications and also an SDK that can be used to develop new applications, if needed. The SDK provides wrappers for dangerous operations and passes them through the secure container application. Divide [34] and AT&T [35] use package rewriting technique to wrap up dangerous instructions of third-party applications, so that the interaction of these rewritten applications with the outer world happens through the secure container.

3.2.2 Mobile Virtualization

Virtualization provides environments that are isolated from each other, and that are indistinguishable from the "bare" hardware, from the OS point of view. The hypervisor is responsible for guaranteeing such isolation and for coordinating the activities of the virtual machines. Virtualization has been widely used in traditional computers because it can: (i) increase security, and (ii) reduce the cost of deployment of applications (the hardware is shared in a secure way). With the spreading of mobile devices and with the increase of their performance capabilities the question of porting virtualization to mobile platforms became actual. Virtualization for mobile systems provides specific advantages like: (i) the possibility to separate communication subsystems (backed by real-time operating system) from high-level application code (which requires functional rich operating system with good interfaces); (ii) an opportunity to provide licence separation; (iii) a chance to increase the security of the communication stack [36].

However, there are still several barriers for the adoption of virtualization in mobile devices. The main one is that ARM architecture, which is the most popular architecture for mobile devices, has a non-virtualisable instruction set

architecture [7] (except Cortex-A15 design [37], which adds hardware-assisted virtualization capabilities). So as efficiency is a major concern in embedded virtualization, full virtualization approaches (emulation and binary translation) are not yet applicable for these devices because they are computational expensive. Thus, for embedded devices paravirtualization is used, which requires source-code modification of guest operating system [38]. There are several approaches to port popular Linux hypervisors to ARM architecture: Xen [38], L4 [7], KVM [39]. There are also several industry solutions: MVP by VMware [40], OKL4 by OK Labs [41] and vLogix Mobile by Red Bend [42]. All these solutions can be applied to create separate secure environments for business and private use. However, since all these virtual machines are simply ported to mobile platforms while being designed for PCs, they all share low performance.

A much better approach is Cells [9]. Cells is a new virtual machine specifically designed for mobile platforms. It provides lightweight virtualization for Android. The authors modified Android system in such a way that it is possible to have several separated environments, called Virtual Phones, based on the same operating system. Virtual Phones are completely separated from each other using kernel-level and user-level device namespace mechanism.

Yet, a common drawback to all the above solutions is that switching between virtual environments requires user interactions, and the configuration of each virtual environment is hardcoded and cannot be changed by the end-user.

3.2.3 Other Approaches

Besides the above approaches there few other solutions. Gupta et al. [43] modified Android framework to support dual mode of operation, private and enterprise. The modification allows to restrict the use of communication capabilities of a phone, to force communication through enterprise VPN and have an encrypted external storage in enterprise mode. The authors of TrustDroid [44] proposed to monitor IPC communications, network traffic and filesystem access to separate data exchange between different domains, for instance, between enterprise and personal environments.

Other solutions use the capabilities of Taintdroid to track sensitive information. The main difference of these approaches is how to discover sensitive information. For instance, Feth and Jung [21] proposed to rely on external authorities which supply data-usage policies with data. Thus, what data are sensitive in a smartphone is defined by external trusted authorities. In [22], the authors taint all the data that is produced or accessed by enterprise applications as sensitive information. Meanwhile, Ahmed and Ahamad [23] relies on the separation of public and private sources of data to detect sensitive information. Differently from MOSES, none of these solutions detects when a profile is active without user interaction. Furthermore, all of them offer only profiles predefined by the solution developers.

4 MOSES Overview

This section provides an overview of our approach named MOde-of-uses SEparation in Smartphones (MOSES).

MOSES provides an abstraction for separating data and apps dedicated to different contexts that are installed in a single device. For instance, corporate data and apps can be separated from personal data and apps within a single device. Our approach provides *compartments* where data and apps are stored. MOSES enforcement mechanism guarantees data and apps within a compartment are isolated from others compartments' data and apps. These compartments are called *Security Profiles* in MOSES. Generally speaking, a *SP* is a set of policies that regulates what applications can be executed and what data can be accessed.

One of the features introduced in MOSES is the automatic activation of *SP* depending on the context, in which the device is being used. *SPs* are associated with one or more definitions of *Context*. A context definition is a Boolean expression defined over any information that can be obtained from the smartphone's *raw sensors* (e.g., GPS sensor) and *logical sensors*. Logical sensors are functions which combine raw data from physical sensors to capture specific user behaviours (such as detecting whether the user is running). When a context definition evaluates to true, the *SP* associated with such a context is activated. It is a possible situation when several contexts, which are associated with different *SPs*, may be active at the same time. To resolve such conflicts, each *SP* is also assigned with a priority allowing MOSES to activate the *SP* with the highest priority. If *SPs* have the same priority, the *SP*, which has been activated first, will remain active.

MOSES permits a user to manually switch to a specific *SP*. To this end, MOSES provides a system app that the user can employ for forcing MOSES to activate a given *SP*. However, this behaviour can be restricted to avoid that the user activates unwanted *SP* in a given context (for instance, switching to a personal *SP* when at work).

Each *SP* is associated with an owner of the profile and can be protected with a password. A *SP* can be created/edited locally through an app installed on the device. Additionally, MOSES supports remote *SP* management. The former possibility may be used by a user of the phone for managing her personal *SP*, while the latter may be employed by an enterprise administrator to control the work *SP*. To avoid that the user tampers with the work *SP*, the security administrator protects the work *SP* with a password. In this way, MOSES can be used for realising a Mobile Device Management solution to manage remotely the security settings of a fleet of mobile devices.

The current version of MOSES leverages the same idea of lightweight separation of *SPs* as the one presented in [45]. At the same time, although the same idea is exploited, the approach used by MOSES is completely new. The previous version of MOSES [45] completely relies on Taintdroid to split data between different profiles. Data separation occurred using user-defined policies, which restricted the flow of information between different profiles. In the current version of MOSES, the separation of application data is implemented on the Linux kernel level through filesystem virtualization approach. This allows our system to provide app data segregation out of the box. Moreover, a user in the new version of MOSES needs to define security policies only if she wants to apply fine-grained constraints to data.

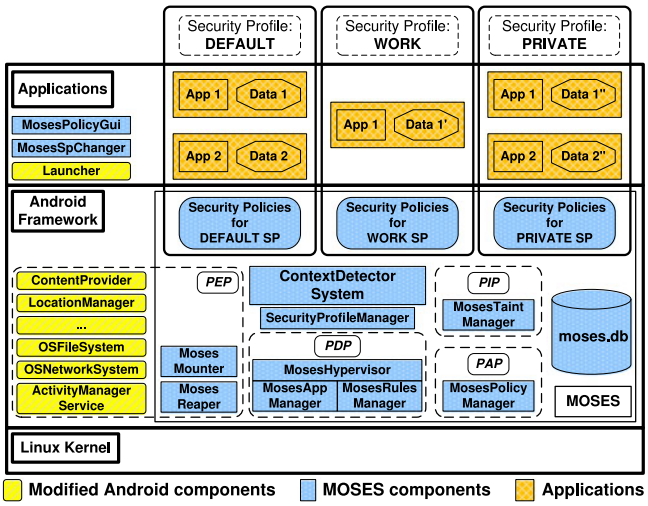

Fig. 1. MOSES architecture.

5 Architecture

MOSES consists of the components presented in Fig. 1. Central to MOSES is the notion of *Context*. The component *ContextDetectorSystem* is responsible for detecting context activation/deactivation. When such an event happens, the *ContextDetectorSystem* sends a notification about this to the *SecurityProfileManager*.

The *SecurityProfileManager* holds the information linking a *SP* with one or more *Context*. The *SecurityProfileManager* is responsible for the activation and deactivation of *SPs*. The *SecurityProfileManager* implements the following logic:

- If a newly activated *Context* corresponds to the active *SP* then the notification is ignored;
- If the *SP* corresponding to a newly active *Context* has a lower or equal priority to the currently running *SP*, then the notification is ignored;
- In all other cases, a *SP* switch has to be performed. This means that the currently running *SP* has to be deactivated and the new *SP* becomes active.

In the latter case, the *SecurityProfileManager* sends a command to the *MosesHypervisor* informing which is the new *SPs* that needs to be activated.

The *MosesHypervisor* is the component that acts as a policy decision point (PDP) in MOSES. The *MosesHypervisor* provides a central point for MOSES security checks against the policies defined for the active *SP* to regulate access to resources. The *MosesHypervisor* delegates the policy checks to its two managers: the *MosesAppManager* and the *MosesRulesManager*. The former is responsible for deciding which apps are allowed to be executed within a *SP*. The latter takes care of managing *Special Rules*.

The *MosesPolicyManager* acts as the policy administrator point (PAP) in MOSES. It provides the API for creating, updating and deleting MOSES policies. It also allows a user to define, modify, remove monitored *Contexts* and assign them to *SPs*. Moreover, this component also controls access to MOSES policy database (`moses.db`) allowing only applications with special permissions to interact with this component.

The *MosesTaintManager* component manages the "shadow database" which stores the taint values used by Taintdroid. We have extended the functionality of Taintdroid to perform more fine-grained tainting. In MOSES, we can taint specific rows of a content provider: to be able to perform per row filtering when an app access data in the content provider. For instance, it is possible to filter out from the query result data the rows which contain the information about device identifiers or user contacts. Given the fact that the enforcement of policies depends on the information provided by the *MosesTaintManager*, this component acts as a policy information point (PIP).

The decisions taken by the *MosesHypervisor* need to be enforced by the policy enforcement point (PEP). MOSES affects several components within Android middleware where decisions need to be enforced. For this reason, the PEP includes several Android components offering system services such as *LocationManager* and *ActivityManagerService*. Moreover, some Android core classes (such as the `OSFileSystem` and `OSNetworkSystem`) are modified to enforce decisions regarding the access to the filesystem and network, respectively.

The enforcement of separated *SPs* requires special components to manage application processes and filesystem views. When a new *SP* is activated, it might deny the execution of some applications allowed in the previous profile. If these applications are running during the profile switch, then we need to stop their processes. The *MosesReaper* is the component responsible for shutting down processes of applications no longer allowed in the new *SP* after the switch. In MOSES, applications have access to different data depending on the active profile. To separate data between profiles different filesystem view are supported. This functionality is provided by the *MosesMounter*. More details are considered in Section 6.2.

To allow the user of the device to interact with MOSES, we provide two MOSES applications: the *MosesSpChanger* and the *MosesPolicyGui*. The *MosesSpChanger* allows the user to manually activate a *SP*. It communicates with the *MosesHypervisor* and sends it a signal to switch to the profile required by the user. The *MosesPolicyGui* allows the user to manage *SPs*. We consider this component in details in Section 6.5.

6 Implementation

This section describes implementation details of some key aspects of MOSES. In particular, the version described here is based on the Android Open Source Project (AOSP) [46] version 2.3.4_r1. Moreover, MOSES incorporates the functionality of Taintdroid [4] to taint sensitive data.

6.1 Context Detection

One of the contributions of MOSES is that it can automatically switch *SPs* based on the current *Context*. The *ContextDetectorSystem* is responsible for monitoring *Context* definitions and for notifying the listeners about the activation or deactivation of a *Context*. The *SecurityProfileManager* component, which is one of these listeners, is notified about the change through the callback functions `onTrue(context_id)` and `onFalse(context_id)`, which correspond to activation and deactivation of a *Context* respectively. The `context_id` parameter represents a *Context*

identifier. So as MOSES context detection functionality is decoupled from the rest of the system, it may be easily extended by integrating other context detection solutions [47], [48].

When the system starts up, MOSES selects from the database information about all *Contexts* and corresponding *SPs*. MOSES preserves this information in a runtime map in the form of $\langle C_i, (SP_k, prt_k)_i \rangle$, where C_i is the identifier of *Context* and $(SP_k, prt_k)_i$ is a tuple, which corresponds to the *Context* C_i and consists of *SP* identifier SP_k and the priority prt_k that corresponds to this profile. When the *ContextDetectorSystem* detects that a *Context* C_i becomes active (meaning the *Context* definition is evaluated to true), we select from this map the corresponding tuple $(SP_k, prt_k)_i$ and put it in the list of active *SPs*. Because more than one *Contexts* might be active at the same time, there may be more than one *SP* to switch to. In this case, from the list of active *SPs* the one with the highest priority is selected. If the selected *SP* identifier differs from the identifier of the currently running *SP*, the *ContextDetectorSystem* sends a signal to the *MosesHypervisor* to switch to the new profile. Similarly, when *ContextDetectorSystem* detects that a *Context* C_i becomes inactive, the tuple $(SP_k, prt_k)_i$ is deleted from the list of active *SPs*. After that the selection procedure of a *SP* with the highest priority is repeated.

6.2 Filesystem Virtualization

To separate data between different *SPs*, we use a technique called *directory polyinstantiation* [49]. A polyinstantiated directory is a directory that provides a different instances of itself according to some system parameters. In brief, for each *SP* MOSES creates a separate mount namespace [50].

The Android filesystem structure is quite stable, i.e., the system forces an application to store its files in the application's "home" directory that is `/data/data/<package_name>/` (`<package_name>` is the package name of the application). During the installation of an application, Android creates this "home" folder and assigns it Linux file permissions to allow only the owner of the directory (in this case the application) to access the data stored in it. To provide applications with different data depending on a currently running *SPs*, polyinstantiation of "data" folder may be used, i.e., for each *SP* a separate mount namespace, which points to different "physical" data folder depending on the identifier of a *SP*, may be created. In MOSES the described approach is used with two modifications. The first modification let the system to store all "physical" data directories under one parent directory (`/data/moses_private/`). The second modification creates the bindings not between the whole data folder and its "physical" counterpart, but bindings for separate application folders. The former modification allows MOSES to control direct access to the "physical" directories, while the latter permits to decrease storage overhead, because the usage of some apps is prohibited in some *SPs*.

The *MosesMounter* component is responsible for providing the above functionality. In particular, it receives the list of applications' package names that are allowed to execute in a *SP*. For each package name, the MOSES system builds the paths to the application "home" directory and to its MOSES "physical" counterpart, using the information of the identifier of a newly activated *SP*. These two paths are passed to the *mosesmounter* native tool. This tool at first checks if MOSES "physical" directory exists. If not, then it creates this folder and copies there the initial application data from the corresponding "home" directory. Then the *mosesmounter* mounts the "physical" directory to a "home" directory using the Linux command `mount(target, mount_point, "none", MS_BIND, NULL)` [50], where `mount_point` corresponds to the path of the "home" directory and `target` corresponds to the path of the "physical" folder. Thus, the "home" directory always contains the initial copy of the application data, which are created by the Android system during the application installation. If a new *SP* is created, these initial data are copied to the "physical" directory providing the application with a fresh copy of its initial data as if the application has just been installed. If this process finishes successfully, the *MosesMounter* stores the name of this package in the list of mounted points. Thus, the process of polyinstantiation is completely transparent for the applications: after the mounting the applications work with the same paths as usual, although these paths point to another "physical" locations. Thus, there is no need to modify the applications to support the separation of data between different *SPs*.

Before switching to a new *SP*, the *MosesMounter* has to unmount all previously mounted points using the values stored in the list of mounted points. Similarly to the mounting, the *MosesMounter* passes the path to a mounted point (from the list of mounted points) to the *mosesmounter* tool, which performs unmounting. During this operation it is possible that some processes hold some files opened. In this case, the `unmount` command will fail. To overcome this problem, MOSES sends a `SIGTERM` signal to the process and repeats the unmounting. If after this the unmounting is still unsuccessful, MOSES will send a `SIGKILL` signal to the process and once again will perform the unmount operation.

6.3 Dynamic Application Activation

Each *SP* is assigned with a list of application *UIDs* that are allowed to be run when this profile is active. As it was discussed in Section 2, each application during the installation receives its own *UID*. MOSES uses these identifiers to control which applications can be activated for each *SP*. It should be mentioned that some packages can share the same *UID*. This happens if the developer of these applications have explicitly assigned the same value to `sharedUserId` property in the manifest files of the applications, and signed these packages with the same certificate. Thus, during the installation of these applications, the Android system assigns them the same *UID*. In this case, MOSES cannot distinguish these applications and if one of them is allowed in one profile the other will be allowed as well.

During the *SP* switching, the *MosesAppManager* selects from the MOSES database the list of *UIDs*, which are allowed in the activated profile, and stores it into the set of allowed *UIDs*. To control the launch of applications' services and activities, the hooks into the

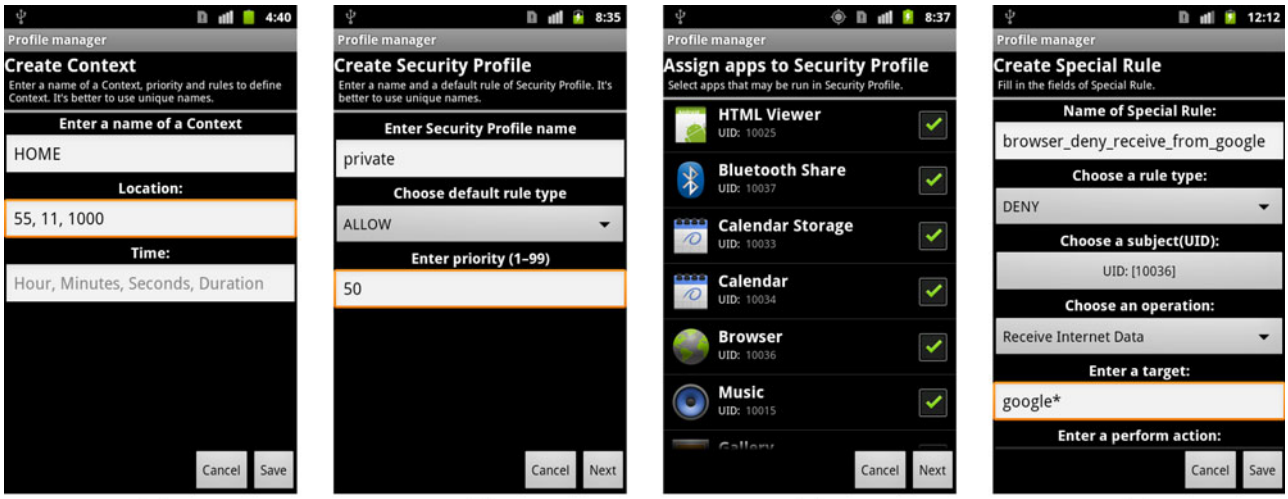

Fig. 2. Screenshots of MOSES Profile Manager application: (a) Context creation, (b) security profile creation, (c) application assignment to a security profile, (d) ABAC rule creation.

`retrieveServiceLocked` and `startActivityMayWait` methods of the `ActivityManagerService` and the `ActivityStack` classes correspondingly are put. These hooks communicate with the *MosesAppManager* and check against the set of allowed apps if a component of an application can be launched. Additionally, the *MosesAppManager* controls the appearance of application icons in Android's Launcher application. When a new *SP* is activated, only the icons of the allowed applications for this profile will be displayed.

6.4 Attribute-Based Policies

Within each *SP*, MOSES enforces an attribute based access control (ABAC) model [51]. The idea is that within each *SP*, users can define fine-grained access control policies to constraint application behaviour. For instance, the user may want to deny an application to read the files on an external storage. In this case, the user may write a policy which will still let the application to run within the profile but the access of this application to files on an external storage will be limited. For defining and editing policies, MOSES provides an activity shown in Fig. 2d.

We have defined a simple policy language using the ABAC model. The attributes types that are taken into consideration in the MOSES language are capitalised in Listing 1. These are *Subject*, *Operation*, *Taint*, *Target*, and *SP-Name*. These attributes are described in the following:

```
1 Subject Operation [Taint] Target
2 decision [perform action(param-list)]  with scope SP-Name
```

Listing 1. Policy language used for ABAC rules

Subject represents the application to which the rule is applied. The application *UID* is retrieved through the GUI provided by MOSES for management (see Fig. 2c), listing the applications installed in the system.

Operation is the action that the subject is executing. The value of this attribute is dependent on the control hooks added by MOSES in the Android framework. Each hook communicates with a special class in the `framework` library that processes the information obtained from the operation hook. For instance, for controlling access to ContentProvider, we have injected hooks in the `ContentResolver` class. Similarly, for network and filesystem operations we have injected hooks in the `core` library. Currently, MOSES supports the following *Operation* types:

- *ContentProvider:* Query ContentProvider, Insert in ContentProvider, Update ContentProvider, Delete ContentProvider.
- *LocationProvider:* Get Last Known Location, Request Location Updates, Add Proximity Alert, Request Single Update.
- *Network:* Receive Internet Data, Send Data to the Internet.
- *Filesystem:* Read from a File, Write to a File.
- *DeviceId:* Get Device ID.

MOSES supports also information flow control using the tainting mechanism provided by Taintdroid. Policies can include the optional attribute *Taint* to specify the taint type associated with the data accessed by the subject.

The *Target* attribute represents the resource that is being accessed. It can have either fixed or volatile values. The values for *LocationProvider* and *DeviceId* operations are fixed and correspond to `GPS` and `IMEI`. In the case of *ContentProvider*, *Network* and *Filesystem* the *Target* values are volatile. This means that a target may be specified partially. For instance, for the *Filesystem* operations the user can specify the following partial target `[/data/data/<package>/*]`. We developed these volatile targets to allow the system to enforce different behaviour for the targets that differs only partially. If a user wants to specify different access behaviour, for instance, for a directory and its subdirectory, she should create a separate policy rule for the directory and another one for the subdirectory.

SP-Name attribute represents the *SP* name where the policy is valid.

The decisions that can be assigned to policy rules are: `ALLOW`, `DENY` and `ALLOW_WITH_PERFORM`. The effects

of the first two are obvious. The decision `ALLOW_WITH_PERFORM` corresponds to allow with a restrictive obligation that performs additional action of the data returned by the operation. For instance, for "Get Device ID" operation a function can be chosen that will obfuscate the real `IMEI` of the device. The functions and their implementations are specified in a special built-in library. `ALLOW_WITH_PERFORM` decision manage to enforce security constraints specified in the profile minimizing the impact of already installed applications.

It is possible that two or more rules may be defined for the same attribute values. To resolve these conflicts, the user should assign a priority value to each rule. In this case, the decision of the rule with the highest priority will have precedence over the decisions of other rules; in the case of equal priorities, then the last inserted rule takes priority.

For some combinations of attribute values, it might be the case when no rules apply. In this case, our system uses a default decision value (either `allow` or `deny`), which is assigned to the *SP*.

6.5 Security Profile Management

To give a user the ability to manage the *SPs* in her device, the *MosesPolicyGui* application is developed. This is a system application signed with a system key and assigned with a special permission. This allows *MosesPolicyGui* application to communicate with the *MosesPolicyManager* and manage the *SPs*. Fig. 2 provides several screenshots of the application running on a device. Due to the lack of space, we will not show screenshots of all activities the application provides.[1]

The *MosesPolicyGui* manages *Contexts* and *SPs*. We develop an application that allows a user to easily configure MOSES functionality. Fig. 2a shows how to create a new *Context* definition. The user specifies the name of a *Context* and the parameters of the sensors used to detect the context around the device.

To define a new *SP* the application provides a wizard that guides the user through the steps. Fig. 2b shows the first activity of this wizard. In this activity, the user has to define the name of a profile, the default decision and the profile priority. The screenshot in Fig. 2c shows how to assign applications to the *SP*. Finally, Fig. 2c shows how to create an ABAC policy rule to deny the browser (UID 10036) to access Google's homepage.

7 MOSES Evaluation

In this section, we report on the thorough experiments we ran to evaluate the performance of MOSES. For all the experiments, we used a Google Nexus S phone.

7.1 Energy Overhead

To measure the energy overhead produced by MOSES, we performed the following tests. We charged the battery of our device to the 100 percent. Then, every 10 minutes we run four system applications (sequentially) via a monkeyrunner [54] script: Calculator, Browser, Contacts and Email. For each of them the script performed common operations

1. The demo presented at [52] is available on our website [53].

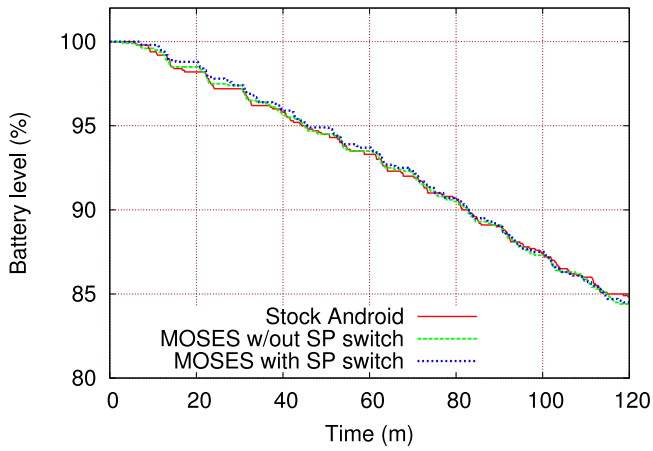

Fig. 3. Energy overhead.

representative for the applications (multiplication of numbers in case of Calculator, browsing several webpages in case of Browser, calling a number and creating an account in case of Contacts, and composing and sending a email in case of Email application). Each experiment lasted for a total of 120 minutes. We executed this experiment for three types of systems: Stock Android, MOSES without *SP* changes, and MOSES with *SP* changes (the system switched between two profiles every 20 minutes).

During each experiment, every 10 seconds, our service measured the level of the battery and wrote this value into a log file. For each of the three considered systems, we executed the test 10 times and averaged the obtained values. The results of this experiment are reported in Fig. 3. We note that the curves for the three considered systems behave similarly. This shows that the fact that MOSES is just running, or even switching between context does not incur a noticeable energy overhead.

7.2 Storage Overhead

One of the most significant overheads produced by MOSES is the storage overhead. In fact, the separation of data for different *SPs* means that some application information will be duplicated in different profiles.

In general, the storage size consumed by a system can be expressed by the following equation:

$$size = size(OS) + \sum_{j=1}^{k} size(AE_j) + \sum_{j=1}^{k} size(AD_j), \qquad (1)$$

where OS is the operating system, AE_j and AD_j, are the application executables and the application data or the jth application. In the specific case of MOSES, $size(AD)$ is equal to:

$$size(AD_{MOSES}) = \sum_{i=1}^{n+1} \sum_{j=1}^{k} (size(AD_{ij})), \qquad (2)$$

where $size(AD_{ij})$ is the size of the data of the jth application in the ith *SP*, k is the number of installed applications, and n is the number of *SPs*. One additional copy of application data (i.e., the $(n+1)$th one) is required to store initial

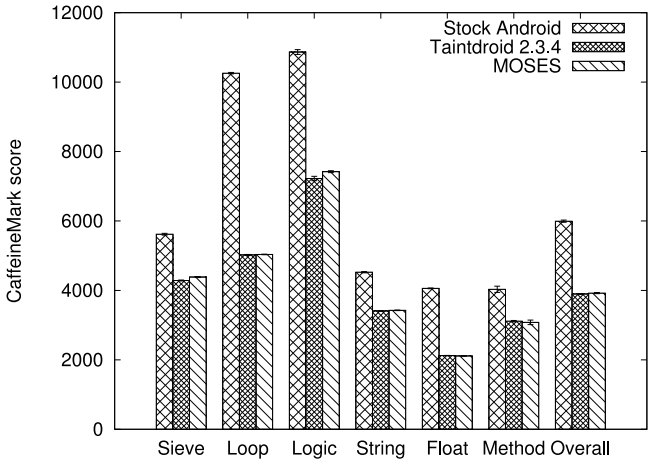

Fig. 4. CaffeineMark Java benchmark results (with standard deviation).

information of all applications. If a new *SP* is created, we need a "clean" copy of application data to be replicated into this new profile. Hence, MOSES stores a copy of application data just after the installation of the application: this copy is later used for replication when a new *SP* is created. It should be mentioned that for MOSES only the initial data of applications are duplicated. The data produced by applications during runtime are not replicated between *SPs*. Second, the data of applications, which are not allowed in a profile, are not copied into the profile.

When comparing MOSES with competitor approaches, MOSES produces less storage overhead. For instance, in case of mobile virtualization [40], [41], [42] not only application data are duplicated (as for MOSES), but also application executables and an operating system (sometimes partially [9]). Dual persona approaches [29], [30], [34], [35] additionally should have a separate copy of application executables in different profiles. Thus, MOSES adds less overhead comparing to this set of approaches because it only works with one copy of application executables.

Moreover, other improvements (currently left as future works) are possible for MOSES, e.g., currently for each *SPs* MOSES stores its own copy of the shared libraries of an application, instead they could be shared among the different profiles.

7.3 Microbenchmark

To assess the overall performance of our system, we decided to run a set of experiments with a benchmarking system. In particular, we used the Java microbenchmark CaffeineMark (version 3.0) [55] ported on the Android platform. This benchmark runs a set of tests which allows a user to assess different aspects of virtual machine performance. The benchmark does not produce absolute values for the tests. Instead it uses internal scoring measures, which are useful only in case of comparison with other systems. The overall score of CaffeineMark 3.0 is a geometric mean of all individual tests. That is why, to assess MOSES we decided to compare it with Stock Android system (version 2.3.4_r1) and Taintdroid system [4] (based on the Android system version 2.3.4_r1). We included in the comparison Taintdroid because MOSES incorporates its functionality, so we wanted to highlight the additional overhead

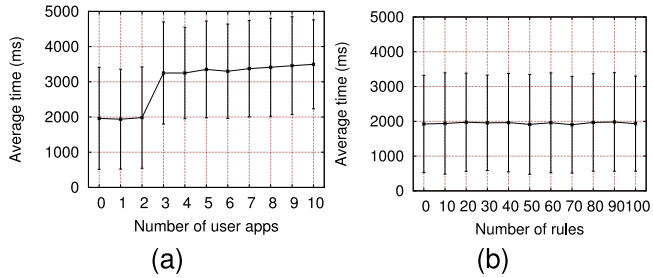

Fig. 5. Time for profile switch (with standard deviation) as a function of the number of: (a) User applications, (b) special rules.

that MOSES introduces compared to Taintdroid. We ran each benchmark 10 times.

For CaffeineMark 3.0 Java benchmark the observed results are reported in Fig. 4. From the figure, we notice that the results for Taintdroid and MOSES are almost the same: the checks that MOSES implements on top of Taintdroid do not have a significant influence on the results of this benchmark. Meanwhile, the difference of overall scores between unmodified (Stock Android) and modified systems (either Taintdroid or MOSES) is quite big. In fact, we can observe the performances are reduced by a 34 percent: the benchmark scores are 5,910.7 for Stock Android, while 3,895.9 for Taintdroid and 3,923.3 for MOSES, the main contributors to this overhead are Loop (about 51 percent overhead) and Float (48 percent) tests.

7.4 Security Profile Switch Overhead

In this section, we present the results of the experiments measuring the time required to switch between *SPs*. We remind that during the profile switch (from an "old" to a "new" profile), MOSES performs the following operations: the unmounting of the data folders of the old profile, the mounting of data folders of the new profile, the unloading of the old and the loading of the new *Special Rules*. Therefore, the time to switch between *SPs* should depend on the number of *Special Rules* and the number of user apps. To find out the dependency between the time and these parameters we ran two sets of experiments. First, we measured the time required to switch *SPs* varying the number of user applications. Then, we did the same measurement while changing the number of *Special Rules*. To measure this time, we put a call `SystemClock.elapsedRealtime()` before and after the switching operations, and calculated the difference between the values produced by this function.

To explore the dependency between the time and the number of applications we varied the number of user applications from 0 to 10. For each number of applications, a clean MOSES system was used (i.e., the system had been flashed on the phone just before the experiment). Then, a *SP* was created allowing all applications to be launched. Then, we measured the time of switch between this new profile and `DEFAULT` *SP*. For each number of applications we repeated the switch for 20 times and then calculated the average time of the switch. For all experiments the same set of 10 applications was used.

The results are shown in Fig. 5a. From this figure, we observe that the switching time increases with the number of applications: moving from 1,962 ms for 0 applications to

TABLE 1
Operation Time: Average (AV), Standard Deviation (SD), Overhead (OV)

SYSTEM	Operations								
	Get Device ID			Query ContentProvider			Write to a File		
	AV, ms	SD, ms	OV, %	AV, ms	SD, ms	OV, %	AV, ms	SD, ms	OV, %
Stock Android	1.018	0.700	0.00	10.516	7.653	0.00	0.818	2.893	0.00
Taintdroid	1.190	0.775	16.90	12.768	8.140	21.41	0.821	2.921	0.37
Moses_000	1.854	0.896	82.12	20.228	8.682	92.35	1.890	2.760	131.05
Moses_010	1.948	0.954	91.36	20.444	9.155	94.41	2.154	2.808	163.33
Moses_020	1.951	0.911	91.65	20.721	9.419	97.04	2.190	2.777	167.73
Moses_030	2.003	1.876	96.76	20.820	8.849	97.98	2.240	2.891	173.84
Moses_040	2.020	0.905	98.43	20.879	9.362	98.55	2.368	2.786	189.49
Moses_050	2.018	0.849	98.23	20.772	9.039	97.53	2.459	2.856	200.61
Moses_060	2.018	0.907	98.23	21.018	9.188	99.87	2.459	2.624	200.61
Moses_070	2.027	1.094	99.12	20.894	8.792	98.69	2.586	3.249	216.14
Moses_080	2.036	0.893	100.00	21.195	9.323	101.55	2.619	2.716	220.17
Moses_090	2.117	0.775	107.96	21.405	9.483	103.55	2.640	2.661	222.74
Moses_100	2.127	0.873	108.94	21.096	8.937	100.61	2.654	2.526	224.45

3,496 ms for 10 applications. The rise of the time is associated with the increase of mounting and unmounting operations that MOSES performs during the switch (see Section 6.2). Furthermore, we note that the time is not uniformly rising with the growth of the number of applications. In particular, after the third application we observe a sharp increase of the function. The explanation of this phenomena is the following. The third application (named `com.antivirus`) after the installation starts a service that opens a file (`google_analytics.db`) and keeps it opened. Thus, MOSES has to kill the service before the unmounting could be performed successfully. In fact, MOSES system is designed in such a way that at first it simply tries to unmount the folder. Then, if this operation is unsuccessful, it sends to the blocking process a `SIGTERM` signal and tries to unmount again. If this try fails then MOSES kills the process, which holds a file opened, and performs the unmounting. Between the different tentatives, MOSES sleeps for 200 ms. We observe that the main time overhead is brought by these unsuccessful unmountings. The spread between three and 10 applications is merely about 250 ms.

The second experiment was conducted similarly to the first one, but in this case we varied the number of *Special Rules* assigned to a new *SP*: from 0 to 100, increasing by 10 rules each time. The results of this experiment are reported in Fig. 5b. As we can see, the time of the switch slowly increases with the increase of the number of rules. We can also note that the standard deviation for the reported values is significant.

We also noticed that the time for the first change of profiles is considerably higher than for the following switches (although this cannot be inferred from the graphics which report average values). For instance, for five applications the time of the first switch is 9,172 ms while for the second is just 3,431 ms. In fact, during the first switch, for each application MOSES has to copy the initial data of an application to a new profile. That is why, the time for the first switch is several times higher than for the following switches. This fact also explains the wide spread of the standard deviation on the graphics.

7.5 Overheads of Fine-Grained Control

As for MOSES fine-grained control overhead, it is mainly due to the checks of ABAC rules. To assess this overhead, we developed three different applications. The first application gets the IMEI of the phone, and stores it into the Android log. The second app reads the information about 10 contacts from the address book of a phone. The third one writes 1,000 characters in a file. We measured the time of the checked operation using system function `SystemClock.elapsedRealtime()`. We ran these applications on different systems: Stock Android, Taintdroid (version 2.3.4) and on several variants of MOSES system, which differ from each other by the number of ABAC rules. In particular, we varied the number of rules for each app from 0 to 100, therefore, the total number of rules in the system changed from 0 to 300 (since there were three different applications). All ABAC rules for an app were the same. An example of ABAC rule for filesystem application (*MsFsTester*) is provided in Listing 2. During the assignment of the rules to *SP*, we assigned the same priority to all of them. We underline that this means considering the worst case, since during the check MOSES has to consider each rule. For each system, we ran each type of check for 1,000 times and calculated the average, standard deviation and overhead for each variant of the systems. The results are summarized in Table 1. *Moses_010* in "System" column means that there are 10 ABAC rules for each considering operation resulting to total 30 rules entered into the system. Similarly, *Moses_000* means that there are no rules in MOSES system assigned to the current *SP*.

```
1 "UID of MsFsTester" "Write to a File" "/data/data/org.
      mosesdroid.msfstester/files/test"
2 "ALLOW" with scope "WORK"
```

Listing 2. Example of ABAC rule

The developed applications represent three different check strategies that are implemented in MOSES. As described in Section 6.4, for each operation which MOSES can check, there is a separate class implemented in the `framework` library. According to the first strategy, the hook for MOSES check is embedded into the `framework` library. In this case, the hook simply calls the check method of the corresponding operation class and enforces the result of the check. For instance, this strategy is used for "Get Device ID" operation check. In the second strategy, the hook is also placed into the `framework` library, but in the corresponding operation

classes the additional checks against *ContentProvider*'s shadow database are performed. A representative of this strategy is the check of "Query ContentProvider" operation. In the third strategy, the hook is placed into the `core` library while the check is performed in the `framework` library. The call of the check method in the corresponding operation class is performed using Java reflection. The strategy is used for "Write to a File" operation check. Thus, *MsImeiTester*, *MsCpTester* and *MsFsTester* were developed to assess the overheads of these three strategies correspondingly.

Comparing the results obtained for Stock Android, Taintdroid and Moses_000, we can see that the main time overheads are added by MOSES operation checks. In fact, Taintdroid adds 16.9 percnt time overhead in case of "Get Device ID" operation, 21.4 percent for "Query ContentProvider" and 0.4 percent in case of "Write to a File" operation, while MOSES even with no ABAC rules adds 82.1, 92.4 and 131.1 percent overheads correspondingly comparing with Stock Android. Further, the results show that the most time consuming operation check is "Write to a File". Not surprisingly, because Java reflection used in the third MOSES check strategy is a quite expensive operation.

As we expected, time overheads grow if we increase the number of rules. We can see that Moses_000 adds 82.1, 92.4 and 131.1 percent overheads. At the same time, Moses_100 adds 108.9, 100.6 and 224.5 percent. We can see that relative overhead for "Get Device ID" operation is higher than for "Query ContentProvider". On the other hand, the absolute overhead is 0.273 ms for the first operation and 0.868 ms for the second operation. Thus, the small percentage in case of the second operation can be explained simply by the fact that it takes more time to process the results of the operation in the test application.

The absolute values of overheads of our three operations between Moses_000 and Moses_100 are 0.273, 0.868 and 0.764 ms, respectively. The difference between the values is connected with the fact that it takes different time to process *target* attribute type in case of fixed and volatile *targets*. In case of fixed *target*, there is no need to compare the attribute value with the pattern. On the contrary, in case of volatile *target* MOSES has to compare the *target* value with the pattern in each relevant ABAC rule.

We assume that in a production system the total number of rules for a *SP* may be higher then the maximum number considered in our experiments (because the number of applications in a production system might be higher). At the same time, the number of rules for a tuple (UID, Operation) is smaller in real-world scenarios than considered in our experiments. So as the tuple (UID, Operation) serves as an index in our system, the number of rule checks in a production system, which causes the main part of the time overhead, will be smaller than the number of rules considered during the experiments. Therefore, we assume that the main part of time overhead in a real-world system will be caused not by the number of ABAC rules in the system but by the embedded MOSES check itself. Unfortunately, this is the price we have to pay providing additional security mechanisms.

8 Conclusions and Future Work

MOSES is the first solution to provide policy-based security containers implemented completely via software. By acting at the system level we prevent applications to be able to bypass our isolation. However, at the present moment MOSES has also some limitations. At first, fine-grained policies and allowed applications are specified using the UID of an application. Meanwhile, in Android it is possible that some applications share the same UID. Thus, if we apply MOSES rules and restrictions to one application they automatically will be extended to the other ones with same UID. Furthermore, some fine-grained policies in MOSES are built on top of Taintdroid [4] functionality. Thus, MOSES inherits the limitations of Taintdroid explained in Section 3. It should be also mentioned that the applications that have root access to the system can bypass MOSES protection. Thus, MOSES is ineffective in combating with the malware that obtains root access, e.g., rootkits.

MOSES can also be improved in several aspects. For instance, to make the policy specification process easier, a solution could be to embed into the system policy templates that can be simply selected and associated to an application. It should be also mentioned that currently MOSES does not separate system data (e.g., system configuration files) and information on SD cards. In the future we plan to add this functionality to the system. Moreover, performance overheads are also planned to be reduced considerably in the future versions.

Acknowledgments

Mauro Conti was supported by a EU Marie Curie Fellowship for the project PRISM-CODE (grant no. PCIG11-GA-2012-321980). This work has been partially supported by the TENACE PRIN Project (grant no. 20103P34XC) funded by the Italian MIUR.

References

[1] Gartner Says Smartphone Sales Accounted for 55 Percent of Overall Mobile Phone Sales in Third Quarter of 2013, http://www.gartner.com/newsroom/id/2623415, 2014.

[2] Are Your Sales Reps Missing Important Sales Opportunities? http://m.sybase.com/files/White_Papers/Solutions_SAP_Reps.pdf, 2014.

[3] Unisys Establishes a Bring Your Own Device (BYOD) Policy, http://www.insecureaboutsecurity.com/2011/03/14/unisys_establishes_a_bring_your_own_device_byod_policy/, 2014.

[4] W. Enck, P. Gilbert, B.-G. Chun, L.P. Cox, J. Jung, P. McDaniel, and A.N. Sheth, "Taintdroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *Proc. Ninth USENIX Conf. Operating Systems Design and Implementation (OSDI '10)*, pp. 1-6, 2010.

[5] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale," *Proc. Fifth Int'l Conf. Trust and Trustworthy Computing (TRUST '12)*, pp. 291-307, 2012.

[6] Y. Xu, F. Bruns, E. Gonzalez, S. Traboulsi, K. Mott, and A. Bilgic, "Performance Evaluation of Para-Virtualization on Modern Mobile Phone Platform," *Proc. Int'l Conf. Computer, Electrical, and Systems Science and Eng. (ICCESSE '10)*, 2010.

[7] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter, "L4Android: A Generic Operating System Framework for Secure Smartphones," *Proc. First ACM Workshop Security and Privacy in Smartphones and Mobile Devices (SPSM '11)*, pp. 39-50, 2011.

[8] T.U. Dresden, and U. of Technology Berlin, "L4Android," http://l4android.org/ 2014.
[9] J. Andrus, C. Dall, A.V. Hof, O. Laadan, and J. Nieh, "Cells: A Virtual Mobile Smartphone Architecture," *Proc. 23rd ACM Symp. Operating Systems Principles (SOSP '11)*, pp. 173-187, 2011.
[10] Android, http://www.android.com/ 2014.
[11] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android Security," *IEEE Security and Privacy*, vol. 7, no. 1, pp. 50-57, Jan./Feb. 2009.
[12] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer, "Google Android: A Comprehensive Security Assessment," *IEEE Security and Privacy*, vol. 8, no. 2, pp. 35-44, Mar./Apr. 2010.
[13] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints," *Proc. Fifth ACM Symp. Information, Computer and Comm. Security (ASIACCS '10)*, pp. 328-332, 2010.
[14] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically Rich Application-Centric Security in Android," *Proc. Ann. Computer Security Applications Conf. (ACSAC '09)*, pp. 73-82, 2009.
[15] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich, "CRêPE: A System for Enforcing Fine-Grained Context-Related Policies on Android," *IEEE Trans. Information Forensics and Security*, vol. 7, no. 5, pp. 1426-1438, Oct. 2012.
[16] A.R. Beresford, A. Rice, and N. Skehin, "MockDroid: Trading Privacy for Application Functionality on Smartphones," *Proc. 12th Workshop Mobile Computing Systems and Applications (HotMobile '11)*, pp. 49-54, 2011.
[17] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh, "Taming Information-Stealing Smartphone Applications (on Android)," *Proc. Fourth Int'l Conf. Trust and Trustworthy Computing (TRUST '11)*, pp. 93-107, 2011.
[18] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These Aren't the Droids You're Looking for': Retroffiting Android to Protect Data from Imperious Applications," *Proc. 18th ACM Conf. Computer and Comm. Security (CCS '11)*, pp. 639-652, 2011.
[19] G. Russello, B. Crispo, E. Fernandes, and Y. Zhauniarovich, "YAASE: Yet Another Android Security Extension," *Proc. IEEE Third Int'l Conf. Social Computing and Privacy, Security, Risk and Trust (SocialCom/PASSAT)*, pp. 1033-1040, 2011.
[20] D. Feth and A. Pretschner, "Flexible Data-Driven Security for Android," *Proc. IEEE Sixth Int'l Conf. Software Security and Reliability (SERE '12)*, pp. 41-50, 2012.
[21] D. Feth and C. Jung, "Context-Aware, Data-Driven Policy Enforcement for Smart Mobile Devices in Business Environments," *Proc. Int'l Conf. Security and Privacy in Mobile Information and Comm. Systems (MobiSec '12)*, pp. 69-80, 2012.
[22] P.B. Kodeswaran, V. Nandakumar, S. Kapoor, P. Kamaraju, A. Joshi, and S. Mukherjea, "Securing Enterprise Data on Smartphones Using Run Time Information Flow Control," *Proc. IEEE 13th Int'l Conf. Mobile Data Management (MDM '12)*, pp. 300-305, 2012.
[23] M. Ahmed and M. Ahamad, "Protecting Health Information on Mobile Devices," *Proc. Second ACM Conf. Data and Application Security and Privacy (CODASPY '12)*, pp. 229-240, 2012.
[24] G. Bai, L. Gu, T. Feng, Y. Guo, and X. Chen, "Context-Aware Usage Control for Android," *Proc. Int'l Conf. Security and Privacy in Comm. Networks (SecureComm '10)*, pp. 326-343, 2010.
[25] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies," *Proc. 22nd USENIX Conf. Security (Security '13)*, 2013.
[26] S. Smalley and R. Craig, "Security Enhanced (SE) Android: Bringing flexible MAC to Android," *Proc. 20th Ann. Network and Distributed System Security Symp. (NDSS '13)*, 2013.
[27] Dual Persona Definition, http://searchconsumerization.techtarget.com/definition/Dual-persona, 2014..
[28] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "AppGuard—Real-Time Policy Enforcement for Third-Party Applications," Technical Report A/02/2012, Saarland Univ., 2012.
[29] J. Jeon, K.K. Micinski, J.A. Vaughan, A. Fogel, N. Reddy, J.S. Foster, and T. Millstein, "Dr. Android and Mr. Hide: Fine-Grained Permissions in Android Applications," *Proc. Second ACM Workshop Security and Privacy in Smartphones and Mobile Devices (SPSM '12)*, pp. 3-14, 2012.
[30] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical Policy Enforcement for Android Applications," *Proc. USENIX 21st USENIX Conf. Security Symp. (Security '12)*, p. 27, 2012.
[31] NitroDesk TouchDown, http://www.nitrodesk.com/TouchDown.aspx, 2014.
[32] Good BYOD Solutions, http://www1.good.com/mobility-management-solutions/bring-your-own-device, 2014.
[33] Fixmo SafeZone: Corporate Data Protection, http://fixmo.com/products/safezone, 2014.
[34] Divide Webpage, http://www.divide.com/, 2014.
[35] AT&T Toggle, https://www.wireless.att.com//businesscenter/solutions/industry-solutions/mobile-productivity-solutions/toggle.jsp, 2014.
[36] G. Heiser, "Virtualization for Embedded Systems," technical report, Open Kernel Labs, Inc., http://www.ok-labs.com/_assets/image_library/virtualization-for-embedded-systems1983.pdf, 2007.
[37] R. Mijat, and A. Nightingale, "Virtualization Is Coming to a Platform Near You," technical report, ARM, http://www.arm.com/files/pdf/System-MMU-Whitepaper-v8.0.pdf.
[38] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim, "Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones," *Proc. IEEE Fifth Consumer Comm. and Networking Conf. (CCNC '08)*, pp. 257-261, 2008.
[39] C. Dall and J. Nieh, "KVM for ARM," *Proc. Ottawa Linux Symp.*, 2010.
[40] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis, "The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket?" *ACM SIGOPS Operating Systems Rev.*, vol. 44, no. 4, pp. 124-135, Dec. 2010.
[41] OKL4 Microvisor, http://www.ok-labs.com/products/okl4-microvisor, 2014.
[42] Mobile Virtualization, http://www.redbend.com/en/products-services/mobile-virtualization, 2014.
[43] A. Gupta, A. Joshi, and G. Pingali, "Enforcing Security Policies in Mobile Devices Using Multiple Personas," *Proc. Seventh Int'l ICST Conf. Mobile and Ubiquitous Systems: Computing, Networking, and Services (MobiQuitous)*, pp. 297-302, 2010.
[44] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, "Practical and Lightweight Domain Isolation on Android," *Proc. First ACM Workshop Security and Privacy in Smartphones and Mobile Devices (SPSM '11)*, pp. 51-62, 2011.
[45] G. Russello, M. Conti, B. Crispo, and E. Fernandes, "MOSES: Supporting Operation Modes on Smartphones," *Proc. 17th ACM Symp. Access Control Models and Technologies (SACMAT '12)*, pp. 3-12, 2012.
[46] Android Open Source Project (AOSP), http://source.android.com/, 2014.
[47] B. van Wissen, N. Palmer, R. Kemp, T. Kielmann, and H. Bal, "ContextDroid: An Expression-Based Context Framework for Android," *Proc. PhoneSense '10*, pp. 1-5, 2010.
[48] D. Kramer, A. Kocurova, S. Oussena, T. Clark, and P. Komisarczuk, "An Extensible, Self Contained, Layered Approach to Context Acquisition," *Proc. Third Int'l Workshop Middleware for Pervasive Mobile and Embedded Computing (M-MPAC '11)*, pp. 6:1-6:7, 2011.
[49] Ubuntu Manuals - PAM Namespaces, http://manpages.ubuntu.com/manpages/maverick/man8/pam_namespace.8.html, 2014..
[50] Mount(2) - Linux Man Page, http://linux.die.net/man/2/mount, 2014.
[51] E. Yuan and J. Tong, "Attributed Based Access Control (ABAC) for Web Services," *Proc. IEEE Int'l Conf. Web Services (ICWS '05)*, pp. 561-569, 2005.
[52] G. Russello, M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich, "Demonstrating the Effectiveness of MOSES for Separation of Execution Modes," *Proc. ACM Conf. Computer and Comm. Security (CCS '12)*, pp. 998-1000, 2012.
[53] MOSES Project, http://mosesdroid.org/, 2014.
[54] Monkeyrunner, http://developer.android.com/tools/help/monkeyrunner_concepts.html, 2014.
[55] CaffeineMark 3.0 Benchmark, http://www.benchmarkhq.ru/cm30/, 2014.

Yury Zhauniarovich received the MSc degree in computer science from the Belarusian State University in 2007. Since 2009, he has been working toward the PhD degree at the University of Trento, Italy. From 2007 to 2009, he was a SAP consultant. His research interests include design, implementation and evaluation of security enhancements of mobile operating systems, runtime security, smartphone application security, and mobile malware.

Giovanni Russello received the MSc (summa cum laude) degree in computer science from the University of Catania, Italy in 2000, and the PhD degree from the Eindhoven University of Technology in 2006. After obtaining the PhD degree, he moved to the Policy Group in the Department of Computing at Imperial College London. He is a senior lecturer at the University of Auckland, New Zealand. His research interests include policy-based security systems, privacy and confidentiality in cloud computing, smartphone security, and applied cryptography. He is a member of the IEEE.

Mauro Conti received the PhD degree from Sapienza University of Rome, Italy, in 2009. After receiving the PhD degree, he was a postdoctoral researcher at Vrije Universiteit Amsterdam, The Netherlands. In 2008, he was a visiting researcher at the Center for Secure Information Systems, George Mason University, Fairfax, VA. From 2011, he has been an assistant professor at the University of Padua, Italy. In the summer of 2012 and 2013 he was a visiting assistant professor at University of California, Irvine, CA. In 2012, he received a Marie Curie Fellowship by the European Commission. In 2013, he received a DAAD fellowship for visiting TU Darmstadt. His main research interest is in the area of security and privacy. In this area, he has published more than 60 papers in international peer-reviewed journals and conferences. He was a panelist at ACM CODASPY 2011, and a general chair for SecureComm 2012 and ACM SACMAT 2013. He is a member of the IEEE.

Bruno Crispo received the MSc degree in computer science from the University of Turin, Italy in 1993, and the PhD degree from the University of Cambridge, United Kingdom in 1999. He is an associate professor at the University of Trento. His main research interests include distributed systems and cloud security, applied cryptography, access control, mobile security and privacy. He has published more than 110 papers in international journals and conferences on security related topics. He is a member of the ACM and a senior member of the IEEE.

Earlence Fernandes graduated with the bachelor's degree in computer engineering from the University of Pune in 2009. Currently, he is working toward the PhD degree at the University of Michigan, Ann Arbor, working in the areas of smartphone security and runtime optimizations for information flow control. He was a scientific programmer in the Systems Security group at Vrije Universiteit Amsterdam until 2012. He is a member of the IEEE.